BUILDING A RECOMMENDATION ENGINE

ANJALI NAGULPALLY, TOM COOK, WILL JOBS, FLORE UZAN

Abstract. Most recommendation systems make recommendations based on either (1) item-item similarity, based on those items' attributes (e.g., movie genre); or (2) users' ratings of those items. The former, called content filtering, recommends items that are similar to those that that a user rated highly. This provides logical recommendations but is unlikely to expose a user to new items outside their preferences that they might also enjoy. The latter, collaborative filtering, makes recommendations based on a matrix of user-item ratings, without regard for any other information about the items or users themselves. This approach has the advantage of exposing a user to new content they may like, but the recommendations it makes may be surprising. Here we develop a hybrid algorithm that combines these two approaches, generating recommendations that are neither too surprising nor staid. The algorithm is used to make movie recommendations based on information about the movies and the available user ratings.

1. Introduction.

Recommendation systems developed rapidly in response to the massive amount of information and choice provided by the internet, to help users quickly find content, such as websites [11], movies and television shows [4], products [12], and romantic partners [8], relevant to their interests. While not the first implementer of such a system, Amazon helped show the power of such a system with its well-known feature "customers who bought items in your shopping cart also bought" [12]. Since then, it has become a core asset of any site that personalizes content for its users. The most common approaches to implementing a recommendation system are content filtering and collaborative filtering. In the remainder of this paper, the methods will be discussed in terms of movie recommendations, but the techniques generalize to other kinds of recommendations.

Content filtering is perhaps the simplest to understand and implement. Given a set of information about each movie, such as title, genre(s), and year of release, the goal is to identify movies in the dataset that are highly similar to that movie in terms of these properties [3]. For a given user, the algorithm recommends movies that are most similar to movies they rated highly. Similarity may be calculated using cosine similarity, which takes as input two movie vectors (each of which captures the available information for each movie) and computes the cosine of the angle between them [1]. The advantage of the content filtering approach is that the recommendations are unsurprising; a friend with a wealth of knowledge about movies could make similar recommendations. However, this approach suffers from "overspecialization" [23]: users will only get recommendations for highly similar movies (e.g., sequels to movies they liked) and will miss out on less similar movies that they might also like. This problem is especially serious for users with a very small number of ratings, or no ratings at all (known as the "cold-start problem" [21]. With such limited data, the algorithm could be modified to recommend the most-watched movies or to encourage the user to explore diverse categories until more data is obtained.

By contrast, collaborative filtering (also known as "social filtering") uses all users' ratings to make inferences about what an individual user might like, without regard for information about the movies themselves [21]. There are several ways this is typically implemented. Early recommender systems often used a "memory-based" algorithm, which calculates similarity between users' sets of ratings in a manner similar to content filtering [21]. Once a group of similar users is identified for a particular user, the system recommends movies that others users rated highly but that this user has not seen. This can also be implemented by focusing on movies, starting with a movie that a user rated highly, and identifying movies that have highly similar sets of ratings among all other users [12]. This approach is straightforward to implement but it cannot make recommendations for new users or movies, and it does not perform well when the data are sparse [15].

Another way to implement collaborative filtering that can handle data sparsity (at the cost of easy interpretability) is "model-based" collaborative filtering [19]. This type of collaborative filtering may use dimensionality reduction techniques like SVD, clustering techniques, or Bayesian belief nets [21]. These techniques impute missing values by using a model created in a separate phase, prior to the prediction process [2]. The model-based approach has two significant advantages over the memory-based approach: space complexity and prediction speed. Generally speaking, the space required by a model is less than the user-movie matrix used for learning [2]. This is advantageous for representing sparse data, which is common to recommendation datasets; users usually rate a small fraction of items in a collection. By reducing the dimensionality of sparse data, the compact model can make predictions using the dense model, and it can do so efficiently [2].

However, this algorithm suffers the cold-start problem in two ways: a new user and a new movie has no ratings, so the algorithm cannot provide new users recommendations, nor can it recommend the new movie to existing users [21]. The algorithm also struggles to recommend movies to users with very unique tastes, also known as "gray sheep" [21].

A natural solution is to simply combine the two approaches into a single recommendation algorithm. There is precedent for this: content-boosted collaborative filtering was shown to be an improvement over approaches using content filtering or collaborative filtering alone [15]. We explore this hybrid approach, using non-negative matrix factorization to deal with the sparsity inherent to most recommendation datasets.

2. Methods.

2.1. Cosine Similarity & Term Frequency-Inverse Document Frequency.

Cosine similarity measures the resemblance of vectors based on their direction and length; in information theory, it is commonly used to compute the similarity of documents in a collection [14]. It thus provides a natural approach to content filtering; cosine similarity can be used to find movies similar to those that a user rated highly.

To calculate cosine similarity, all unique words across the collection of documents are identified, and each document is converted to a vector of unique word frequencies. As a result, all document-vectors are of the same length. Vectors with the most similar word frequencies will have the highest pairwise cosine similarity, with values closer to one. Vectors that are the most dissimilar will have cosine similarity values near zero. [14]. For vectors **A** and **B**, their cosine similarity is calculated:

$$cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}||||\mathbf{B}||}$$

However, the use of "raw" word frequencies poses an issue for measuring document similarity, as it implies that all words are of equal importance. One solution to this issue is a technique called Term Frequency-Inverse Document Frequency (TF-IDF), which is the product of a word's "term frequency" and its "document frequency" [22]. TF-IDF scales down the weights of words that appear across many documents—that is, words with high document frequency.

Term frequency is the number of times a word t appears in a document d. Inverse document frequency is the the scaled inverse fraction of the number of documents in a collection of size N in which a word t appears, and is given by[17]:

$$idf_t = log\left(\frac{N}{1 + |d\epsilon D : t\epsilon d|}\right)$$

Thus, the scaled frequency of word t in document d is given by:

$$tfidf_{t,d} = tf_{t,d} * idf_t$$

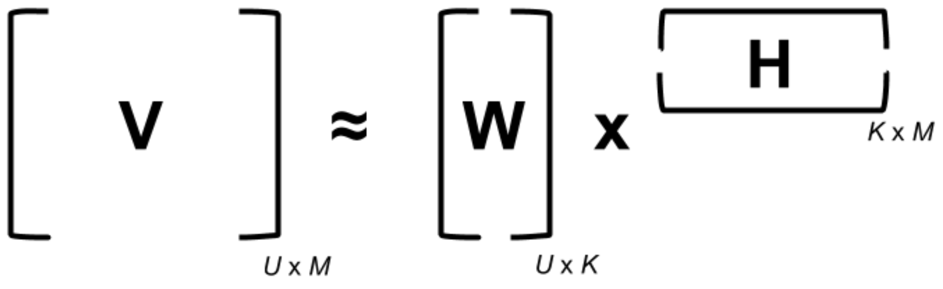
The word frequency vectors are updated with the frequencies from the latter formula, and used to compute the cosine similarity of each pair of documents. The resulting values, ranging from 0 to 1, can be stored in an nxn matrix, with element i, j denoting the pairwise similarity of documents i and j. In the movie recommendation context, employing TF-IDF ensures that judgments of similarity are not based on common descriptive or title words such as "the," "and," or "of."

2.2. Non-Negative Matrix Factorization. Recommendation systems often have to deal with very sparse data, as users consume a small proportion of all items in a collection and rate an even smaller proportion. Further, a user's rating for a particular movie is assumed to be very complicated and dependent on many hidden, unobservable variables. Therefore, one approach to recommendations movies could be to reduce the dimensionality of the data while avoiding overfitting.

Learning objects as a sum of components is likely natural to the human brain [5]. We assume that a user's perception of movies follows this same decomposed representation. That is, a user's rating for a movie can be determined as a linear combination of a movie's features and the user's preference for those features.

Non-negative matrix factorization (NMF) is an algorithm that can learn parts of more complex objects, and is therefore suitable for predicting a user's rating for a movie. NMF factors a large sparse matrix into two dense matrices of rank k (where $k << n$ or m), whose product approximates the large matrix [9].

Fig. 2.1. *Schematic of non-negative matrix factorization*



Each row in matrix **W** represents a user's weights for each of the k features. Each column in matrix **H** represents a movie's weights for each of the k features. The dot product of a user's row in **W** and a movie's column in **H** gives the predicted rating of the user for that movie. By placing non-negative constraints on the values in **W** and **H**, NMF creates a parts-based representation that only allows additive combinations of features without forcing a substantial weight for irrelevant features [9].

NMF approximates the factorization of $\mathbf{V}$. The algorithm begins by initializing the $\mathbf{W}$ and $\mathbf{H}$ matrices, typically assigning random numbers over a specified range. The product of the two matrices is then computed, generating predicted scores for all user-movie pairs, regardless of whether or not an actual rating exists. The error is computed by comparing the predicted score and actual score on user-movie pairs that have actual ratings. The weights in $\mathbf{W}$ and $\mathbf{H}$ are updated based on this error. This process of computing error and updating weight has great implications for implementation.

The objective function considered for optimization in NMF minimizes the Frobenius norm (2.1) between the actual data and the constructed product of the lower rank matrices [10]. Other objective functions, such as Kullback-Leibler divergence, may be used, but are considered beyond the scope of this work.

$$E = \frac{1}{2} \sum_{i,j \in V} (V_{i,j} - W_i \cdot H_j) \tag{2.1}$$

For a sparse matrix, it is impossible to calculate this at every term. The error is computed only on entries that have a rating, while entries that do not have ratings are skipped. Minimization of this objective function can take on various forms. The most intuitive approach is the use of gradient descent. Updating values would take the form, where $E = V - WH$:

$$W_{new} = W + \alpha EH \tag{2.2}$$

$$H_{new} = H + \alpha EW \tag{2.3}$$

Instead of adjusting all values in the matrix at once, a stochastic gradient descent method can be used to update components using error computed for an individual user-movie pair. This is a preferable method for large computations, although convergence may not be as smooth [2]. Updating values is done as follows:

$$\overline{w}_{i(new)} = \overline{w_i} + \alpha e_{i,j} \overline{h_j} \tag{2.4}$$

$$\overline{h}_{i(new)} = \overline{h_i} + \alpha e_{i,j} \overline{w_j} \tag{2.5}$$

The learning rate parameter α has significant implications on the values of the low rank matrices $\mathbf{W}$ and $\mathbf{H}$ [9]. A larger learning rate will create larger corrections, risking the possibility of overshooting a local minimum. A smaller learning rate will increase the computational time learning the model, but will ensure a more stable convergence [2]. In the above methods adhering to the non-negative constraints is tedious. This is typically done by setting negative terms to zero.

Lee and Seung (2000)[10] implement a multiplicative update rule (2.6)(2.7) which changes the learning parameter to eliminate the negative term. Doing so ensures a positive rescaling of the previous values. By forcing a positive rescaling, the values in the $\mathbf{W}$ and $\mathbf{H}$ matrix will be non-negative. Lee and Seung go on to prove that this method ensures convergence for minimizing the cost function despite being a large step size.

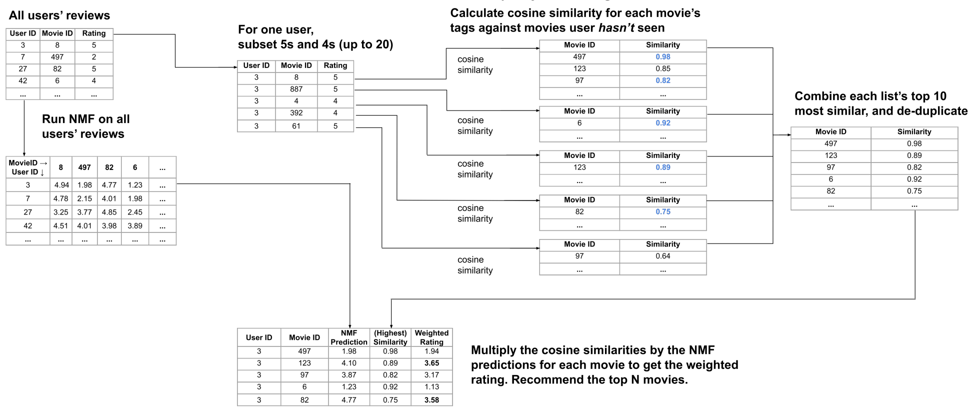
$$w_{i,j} = \frac{(V * H)_{i,j} w_{i,j}}{(WH^T W)_{i,j}} \tag{2.6}$$

$$h_{i,j} = \frac{(V * W)_{i,j} h_{i,j}}{(HW^T H)_{i,j}} \tag{2.7}$$

The multiplicative update rule has been found to provide a good balance of computational speed and ease of implementation [10]. Although the constraints applied are rather simple, the implications of these constraints align with our beliefs regarding the parts-based representation used by the user's brain.

2.3. Hybrid NMF-Cosine Algorithm. A reasonable approach then is to make movie recommendations that combine user ratings and information about the movies. Our algorithm combines using cosine similarity to calculate the similarity between movies in the dataset (based on user tags, genres, titles), and non-negative matrix factorization to reduce the dimensionality of the dataset and thereby impute missing values. A schematic of the algorithm is shown in Figure 2.2 below, and the Python module we developed that implements it is given in Appendix A. Appendix B shows how our module is used to make recommendations for an individual user.

Fig. 2.2. *Schematic of hybrid algorithm*


All users' reviews
Run NMF on all users' reviews
For one user, subset 5s and 4s (up to 20)
Calculate cosine similarity for each movie's tags against movies user *hasn't* seen
cosine similarity
cosine similarity
cosine similarity
cosine similarity
cosine similarity
Combine each list's top 10 most similar, and de-duplicate
Multiply the cosine similarities by the NMF predictions for each movie to get the weighted rating. Recommend the top N movies.

User ID	Movie ID	Rating
3	8	5
7	497	2
27	82	5
42	6	4
...	...	...

MovieID → User ID ↓	8	497	82	6	...
3	4.94	1.98	4.77	1.23	...
7	4.78	2.15	4.01	1.98	...
27	3.25	3.77	4.85	2.45	...
42	4.51	4.01	3.98	3.89	...
...	...	...	...	...	...

User ID	Movie ID	Rating
3	8	5
3	887	5
3	4	4
3	392	4
3	61	5

Movie ID	Similarity
497	**0.98**
123	0.85
97	**0.82**
...	...

Movie ID	Similarity
6	**0.92**
...	...

Movie ID	Similarity
123	**0.89**
...	...

Movie ID	Similarity
82	**0.75**
...	...

Movie ID	Similarity
97	0.64
...	...

Movie ID	Similarity
497	0.98
123	0.89
97	0.82
6	0.92
82	0.75
...	...

User ID	Movie ID	NMF Prediction	(Highest) Similarity	Weighted Rating
3	497	1.98	0.98	1.94
3	123	4.10	0.89	**3.65**
3	97	3.87	0.82	3.17
3	6	1.23	0.92	1.13
3	82	4.77	0.75	**3.58**

First, NMF is computed on the user-movie ratings matrix to calculate an NMF-predicted rating for every user-movie pair in the dataset. This is done once for all users, but must be updated periodically as users rate more movies, and when new movies or users are added. Then, for a specific user, the algorithm selects up to twenty of the user's highly rated movies (rated 4 or 5). Cosine-similarity is used to compare each of these highly-rated movies to every other movie in the dataset based on the movies' genre(s), user-supplied tags, and movie title. This too could be calculated in advance, generating a similarity score for every pair of movies.

Next, for each of the highly rated movies, ten movies which the user has not seen and that have the highest cosine similarity score are combined into a single list. Duplicates are removed from the list, retaining the highest similarity score achieved. For example, if two of the highly-rated movies are *Toy Story* and *Monsters, Inc.*, both may be highly similar to *Toy Story 2*, with cosine similarity scores of 0.98 and 0.8, respectively. Assuming the user has not already seen *Toy Story 2*, and assuming these similarity scores are in the top ten for both of these movies, *Toy Story 2* would be added to the list twice. Then, when duplicates are removed, *Toy Story 2* would have one entry in the list with a score of 0.98. This approach was chosen in order to reflect the maximum similarity a movie achieves to any of a user's highly rated movies.

Finally, this list of movies that are highly similar to movies the user rated highly is combined with the NMF predictions. Each highly similar movie has a cosine similarity score from 0 to 1; these are treated as weights, and are multiplied by the

NMF predicted rating to get a weighted rating that combines content filtering and collaborative filtering to take advantage of all available data.

3. Results. In the sections below we compare the algorithms' recommendations for user ID 37 in our dataset, whose highest rated movies are shown in Table 3.1 below.

TABLE 3.1
Highest ratings of User ID 37

name	rating
Frequency	5
Gladiator	5
Mission: Impossible 2	5
Patriot, The	5
Titanic	5
U-571	5
Butterfly (La Lengua de las Mariposas)	4
Dinosaur	4
Gone in 60 Seconds	4
Perfect Storm, The	4
Terminator, The	4
X-Men	4

3.1. Cosine Similarity. At first, as the initial dataset from Netflix provided only movie titles and genres, cosine similarity was computed using only movie titles. That is, movies were ranked only based on the similarity of their titles. This was not an optimal approach, as movies with similar titles may be quite dissimilar in content. For example, under this approach, *Toy Story*, an animated children's movie about toys that come to life, was deemed highly similar to *West Side Story*, a romantic musical about rival gangs in New York. Further, there were no movies with non-zero cosine similarity to *Jumanji*, due to the film's unique title.

Therefore, we integrated an external dataset from MovieLens — in particular, its set of movie tags. These tags are short, user-generated phrases that describe a movie, such as "murder mystery" or "John Travolta." Matching on both the tags and genres was fruitful, and provided effective, content-based recommendations. This approach was able to pick up on sequels without using titles.

We used this approach of computing cosine similarity on both tags and genres to provide movie recommendations. For each of the movies that a user rated highly, we returned the movies that the user had not yet watched with the highest cosine similarity. Table 3.2 shows the recommendations under this approach for user ID 37, and illustrates the effectiveness of this approach for content-based filtering.

However, content-based filtering only recommends movies similar to those that a user has already seen. That is, it does not "branch out" to movies outside a user's comfort zone, that other users with similar preferences may have liked. Therefore, we would like to integrate this approach with one that improves on the current lack of diversity of recommendations.

3.2. Non-Negative Matrix Factorization. The Netflix dataset contains 31,620 ratings, spanning over 2,353 users and 1,465 movies. With a sparsity of 99.1%, im-

Table 3.2
Cosine similarity recommendations for User ID 37

name	similar_to	similarity
Boat, The (Das Boot)	U-571	0.728223
Back to the Future	Frequency	0.713025
Braveheart	Patriot, The	0.701565
Somewhere in Time	Frequency	0.693946
Tequila Sunrise	Patriot, The	0.681203
Back to the Future Part II	Frequency	0.675175
Forever Young	Patriot, The	0.648162
Mission: Impossible	Mission: Impossible 2	0.645539
Back to the Future Part III	Frequency	0.627702
Tora! Tora! Tora!	U-571	0.618183

puting missing data proved to be challenging. Data may be missing due to different mechanisms [18]. In regards to movie ratings, a user may be actively choosing not to watch a movie, or they may have not had the time to watch the movie. Replacing all missing values with a single value ignores this knowledge.

Initially, the *scikit-learn* decomposition module's implementation of NMF was used, but it required that missing ratings be filled in with a numerical value [16]. Replacing missing values with a numeric value has a high impact on the model's predictions; early tests showed that populating missing values with zero resulted in predictions close to zero. Clearly, this is unacceptable in a recommendation algorithm.

As a result, we used the *Surprise* package which is designed for recommendation algorithms and which calculates the decomposition matrices without any assumptions about the missing values [7].

Recommendations for user ID 37 using NMF alone are shown in Table 3.3. Some of the recommendations are surprising. For example, *Koyaanisqatsi*, an experimental movie consisting of slow motion footage of natural landscapes, is very different from the rest of the top ratings for this user. Another unexpected recommendation is *Les Visiteurs*, a French fantasy comedy. In comparison to the result obtained with cosine similarity, the recommendations from NMF alone are novel, but perhaps surprising to a user. It is possible that these recommendations would be perceived as low quality by the user.

3.3. Hybrid NMF-Cosine Algorithm. Using our algorithm, the top recommendations require both high NMF predictions as well as high similarity to a movie that the user previously enjoyed. The goal is to provide users a mixture of familiarity and novelty, by recommending content they might not otherwise explore, while at the same time recommending movies they may have heard of. Cosine similarity satisfies the familiarity requirement, perhaps too well: many of the recommendations are sequels and movies with the same actor. These may be highly relevant to the users' interests, but may become boring over time. By contrast, NMF successfully exposes the user to new content (e.g., the movie *Koyaanisqatsi* for user ID 37), but might be so unfamiliar that a user does not trust the recommendation system.

Table 3.4 shows the recommendations made by our hybrid NMF-Cosine algorithm for user ID 37. Note that by combining content filtering and collaborative filtering, our algorithm is able to provide diagnostic information about why the recommendations

TABLE 3.3
NMF-based recommendations for User ID 37

name	nmf_prediction
Thing, The	5.000000
Dark City	5.000000
Frighteners, The	5.000000
Beetlejuice	5.000000
Boy Who Could Fly, The	5.000000
Heavenly Creatures	5.000000
Koyaanisqatsi	5.000000
Devil's Advocate, The	5.000000
Peter's Friends	5.000000
My Bodyguard	4.961142
Affair to Remember, An	4.950885
Arachnophobia	4.911703
Visitors, The (Les Visiteurs)	4.895357
Full Monty, The	4.890140
Cyrano de Bergerac	4.879859
Melvin and Howard	4.866165
Children of Heaven, The (Bacheha-Ye Aseman)	4.858580
Peeping Tom	4.748756
Braveheart	4.742437
Fast Times at Ridgemont High	4.720364

TABLE 3.4
Recommendations for User ID 37 based on hybrid approach

name	similar_to	similarity	nmf_prediction	weighted_rating
Braveheart	Patriot, The	0.701565	4.742437	3.327126
Terminator 2: Judgment Day	Terminator, The	0.903422	3.482558	3.146219
Eraser	Terminator, The	0.684168	4.107573	2.810272
Guns of Navarone, The	U-571	0.596033	4.095199	2.440875
Superman II	X-Men	0.586295	4.101303	2.404574
Patton	U-571	0.525239	4.425239	2.324308
Somewhere in Time	Frequency	0.693946	3.343376	2.320121
Superman	X-Men	0.727378	3.151486	2.292320
Bridge on the River Kwai, The	U-571	0.514111	4.435623	2.280403
Back to the Future	Frequency	0.713025	3.128370	2.230605
Boat, The (Das Boot)	U-571	0.728223	2.953605	2.150883
Beauty and the Beast	Dinosaur	0.535022	3.972083	2.125151

are made. Of the 12 ratings on this list, four of them were also provided by the cosine similarity alone. However, *The Boat* and *Back to the Future*, cosine similarity's top two recommendations, appear towards the bottom of this list due to their low NMF predictions. In addition, while *Back to the Future* appears on this list, its two sequels do not, suggesting that our algorithm successfully balances movie similarity with information provided by other users about subjective taste.

Given that the starting point of these recommendations is their cosine similarity to the user's previously rated movies, it is not surprising that these recommendations hew closer to those recommendations than the NMF-alone recommendations. As a result, this list does not contain *Koyaanisqatsi* or *Cyrano de Bergerac*, both of which might be surprising to this user. Encouragingly, this algorithm appears to re-order NMF predictions to take into account similarity to the user's rating history. For example, while *Eraser* has an NMF prediction that is over half a star greater than *Terminator 2*, because the user has already seen and liked *The Terminator*, this algorithm incorporates the similarity into the weighted rating and recommends *Terminator 2* before *Eraser*. Similarly, while *The Bridge on the River Kwai* has the second highest NMF prediction in the list below, it has the lowest similarity to the user's highly rated movies and so appears towards the bottom of this list.

Note that there are some important differences between the movies evaluated by this algorithm compared to those evaluated by the algorithms using either cosine similarity or NMF alone. First, in our implementation of the cosine similarity algorithm, only movies with a 5-star rating were used because it was unclear how a movie with high similarity to a 4-star movie compares to a movie with moderate similarity to a 5-star movie, in terms of a user's preferences. Therefore, it was decided that only the highest-rated movies would be considered. However, in our algorithm, we began with the subset of movies that had either 4- or 5-star ratings. Second, whereas the algorithm using NMF alone considered *all* movies and ranked them by their NMF prediction, NMF predictions were only considered for the movies most similar to the user's highly rated movies. Future work could evaluate whether using all movies in both steps (cosine similarity and NMF predictions) would result in better predictions.

Unfortunately, it is difficult to validate this algorithm objectively. While NMF seeks to minimize the reconstruction error of the ratings matrix, here we impose weights on those predictions, by definition shifting the predictions away from that minimized error. As a result, our recommendations will generally be lower than the actual ratings (since similarity scores are usually less than 1), and the rank ordering of ratings may also be different. However, for the above reasons we are confident that our algorithm adds value to a recommendation system. Our model could be validated in a production system by using an A/B test which recommends movies using our algorithm to one set of users, and either the cosine similarity of NMF algorithm to another set of users, to compare the algorithms' performance at correctly ordering movie preferences.

4. Conclusion. We found that a hybrid algorithm that integrates both content and collaborative filtering provided better recommendations than an algorithm based on only one of the two approaches. The hybrid algorithm mitigates the drawbacks of either approach—namely, that content filtering restricts a user to movies within their "comfort zone", and that collaborative filtering can surprise users with unfamiliar content. It may be worth evaluating some design choices, such as starting with the subset of movies the user rated highly, to see if there are other movies the algorithm may surface with moderate similarity to lower ranked movies but high NMF predictions. In addition, it would be worth investigating other kinds of model-based collaborative filtering algorithms such as Bayesian belief nets. Finally, simulations of how recommendations change as new recommendations enter the system would be useful to evaluate the stability of our algorithm's recommendations.

REFERENCES

[1] Adomavicius, G., & Tuzhilin, A. (2005). Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge & Data Engineering*, (6), 734-749.
[2] Aggarwal, C. *Recommender Systems: The Textbook.* Springer, 2016.
[3] Basu, C., Hirsh, H., & Cohen, W. (1998). Recommendation as classification: Using social and content-based information in recommendation. *Aaai/iaai*, 714-720.
[4] Bennett, J., & Lanning, S. (2007). The netflix prize. *Proceedings of KDD cup and workshop*, 35.
[5] Biederman, I. (1987). Recognition-by-components: a theory of human image understanding. *Psychol. Rev.*:94, 115–147.
[6] Harper, F.M., and Konstan, J.A. (2015). The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, 4: 19:1–19:19.
[7] Hug, N. (2017). Surprise, a Python library for recommender systems. `http://surpriselib.com.`
[8] Kunegis, J., Gröner, G., & Gottron, T. (2012). Online dating recommender systems: The split-complex number approach. *Proceedings of the 4th ACM RecSys workshop on Recommender systems and the social web*, 37-44.
[9] Lee, D.D. & Seung, H.S. (1999). Learning the parts of objects by non-negative matrix factorization. *Nature*:401, 788- 791.
[10] Lee, D. D. and Seung, H.S. (2000). Algorithms for non-negative matrix factorization. *Proceedings of the 13th International Conference on Neural Information Processing Systems (NIPS'00)*, T. K. Leen, T. G. Dietterich, and V. Tresp (Eds.). MIT Press, Cambridge, MA, USA, 535-541.
[11] Lieberman, H., Van Dyke, N. W., & Vivacqua, A. S. (1999). Let's browse: a collaborative Web browsing agent. *IUI*: 99, 65-68.
[12] Linden, G., Smith, B., & York, J. (2003). Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*:1, 76-80.
[13] Luo, X., Zhou, M., Xia, Y., & Zhu, Q. (2014). An Efficient Non-Negative Matrix-Factorization-Based Approach to Collaborative Filtering for Recommender Systems. *IEEE Transactions on Industrial Informatics*: 10, 1273-1284.
[14] Manning, C., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval.* Cambridge University Press.
[15] Melville, P., Mooney, R. J., & Nagarajan, R. (2002). Content-boosted collaborative filtering for improved recommendations. *Aaai/iaai*: 23, 187-192.
[16] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. *Journal of machine learning research*: 12(Oct), 2825-2830.
[17] Ramos, J. (2003). Using TF-IDF to Determine Word Relevance in Document Queries. *Proceedings of the First Instructional Conference on Machine Learning*: 242, 133-142.
[18] Rubin, D. (1976). Inference and missing data, *Biometrika*: 63(3), 581–592.
[19] Schafer, J. B., Frankowski, D., Herlocker, J., & Sen, S. (2007). Collaborative filtering recommender systems. *The adaptive web*, 291-324. Springer, Berlin, Germany.
[20] Smith, G., Camp, G., Boyd, E., & LaFrance, J. (2011). *U.S. Patent No. 8,078,615.* Washington, DC: U.S. Patent and Trademark Office.
[21] Su, X., & Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in artificial intelligence.*
[22] Tata, S. & Patel, J. (2007). Estimating the selectivity of tf-idf based cosine similarity predicates. *ACM SIGMOD Record*: 36, 7-12.
[23] Yang, S., Korayem, M., AlJadda, K., Grainger, T., & Natarajan, S. (2017). Combining content-based and collaborative filtering for job recommendation system: A cost-sensitive Statistical Relational Learning approach. *Knowledge-Based Systems*: 136, 37-45.

Appendix A. Python package NMF_Cosine_Recommender.

```
import numpy as np
import pandas as pd
import random
from surprise import Reader, Dataset
from surprise import NMF
from surprise.model_selection import cross_validate
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
```

```python
class NMF_Cosine_Recommender:
    """[summary]
       @author Will Jobs
    """
    def __init__(self, df_users, df_movies, df_ratings, df_movie_lens_tags,
         biased=False):
        """[summary]

        Args:
            df_users ([type]): [description]
            df_movies ([type]): [description]
            df_ratings ([type]): [description]
            df_movie_lens_tags ([type]): [description]
            biased
        """
        self.users = df_users
        self.movies = df_movies
        self.ratings = df_ratings
        self.ml_tags = df_movie_lens_tags
        self.biased = biased
        self.trained_nmf = False
        self.preprocessed = False
        self.trained_cosine = False
        self.cv_score = None
        self.cv_fit_time = None
        self.movies_merged = pd.DataFrame()
        self.nmf_predictions = pd.DataFrame()
        self.tfidf_matrix = None
        self.algo = None
        self.W = None
        self.H = None

    def preprocess_tags(self, verbose=True):
        """[summary]

        Args:
            verbose (bool, optional): [description]. Defaults to True.
            seed ([type], optional): [description]. Defaults to None.
        """
        if self.preprocessed: # only do this once
            return

        if verbose:
            print('Preprocessing tags and movie information...', end='')

        self.ml_tags.rename(columns={'userId':'userID','movieId':'movieID'},
             inplace=True)
        self.ml_tags = self.ml_tags.astype({'tag':str})

        tmp_tags = self.ml_tags.copy()
        tmp_movies = self.movies.copy()

        # replace punctuation in tags (a space), movie name (a space), and
             genres (no space). These will eventually be folded into the tags
             list
        # doing it this way to avoid altering the original tags during
             presentation later
        tmp_tags['new_tag'] = tmp_tags.tag.str.replace(r'[^\w\s]',' ')
        tmp_movies['new_name'] = tmp_movies.name.str.replace(r'[^\w\s]',' ')
        tmp_movies['new_genre1'] = tmp_movies.genre1.str.replace(r'[^\w\s]','')
        tmp_movies['new_genre2'] = tmp_movies.genre2.str.replace(r'[^\w\s]','')
        tmp_movies['new_genre3'] = tmp_movies.genre3.str.replace(r'[^\w\s]','')

        # aggregate all users' tags up per movie
        tags_nostrip = tmp_tags.groupby('movieID').tag.apply('
             '.join).reset_index()
        tags_nostrip.rename(columns={'tag':'tags'}, inplace=True)
        tags_strip = tmp_tags.groupby('movieID').new_tag.apply('
             '.join).reset_index()
        tags_strip = tags_nostrip.merge(tags_strip, on='movieID')

        # merge name, genres, and tags together
        self.movies_merged = tmp_movies.merge(tags_strip, on='movieID',
             how='left')
        self.movies_merged['tags_strip'] = self.movies_merged.apply(lambda x:
             '{} {} {} {} {}'.format(x['new_name'], x['new_genre1'],
             x['new_genre2'] if type(x['new_genre2']) != float else "",
             x['new_genre3'] if type(x['new_genre3']) != float else "",
             x['new_tag']), axis=1)
        self.movies_merged.drop(columns=['new_tag','new_name','new_genre1','new_genre2','new_genre3'],
             inplace=True)

        # merge in the combined tags (with punctuation)
        self.movies = self.movies.merge(tags_nostrip, on='movieID', how='left')

        self.preprocessed = True

        if verbose:
            print('Done')

    def train_cosine_similarity(self, seed=None, verbose=True):
        """[summary]

        Args:
            seed ([type], optional): [description]. Defaults to None.
            verbose (bool, optional): [description]. Defaults to True.

        Raises:
            RuntimeError: [description]
        """
        if not self.preprocessed:
            raise RuntimeError('Cannot train cosine similarity until
                preprocessing is done (via preprocess_tags)')

        if self.trained_cosine: # only do this once
            return
        if seed is not None:
            random.seed(seed)

        vectorizer = TfidfVectorizer(stop_words='english', min_df=3)

        if verbose:
            print('Cosine similarity training...', end='')

        self.tfidf_matrix =
             vectorizer.fit_transform(self.movies_merged['tags_strip'])
        self.trained_cosine = True

        if verbose:
            print('Done')

    def run_nmf(self, n_factors=15, run_cross_validation=True,
         cv_metric='RMSE', seed=None, verbose=True):
        """[summary]

        Args:
            n_factors (int, optional): [description]. Defaults to 15.
            run_cross_validation (bool, optional): [description]. Defaults to
                 True.
            cv_metric (str, optional): [description]. Defaults to 'RMSE'.
            seed ([type], optional): [description]. Defaults to None.
            verbose (bool, optional): [description]. Defaults to True.
        """

        # ratings get clipped from 1 to 5
        reader = Reader(rating_scale=(1.0,5.0))
        data = Dataset.load_from_df(self.ratings, reader)

        # first, calculate CV on a fraction of the dataset
        if run_cross_validation:
            if verbose:
                print('Running cross-validation...', end='')

            if seed is not None:
                random.seed(seed)

            algo = NMF(n_factors=n_factors, biased=self.biased,
                 random_state=seed)
            cv_results = cross_validate(algo, data, measures=['RMSE'], cv=5,
                 verbose=False)
            avg_cv_result = pd.DataFrame.from_dict(cv_results).mean(axis=0)
            self.cv_score = avg_cv_result['test_' + cv_metric.lower()]
            self.cv_fit_time = avg_cv_result['fit_time']

            if verbose:
                print('Done')
                print('Average CV score: {}\nAverage fit time: {}
                     seconds'.format(round(self.cv_score, 4),
                     round(self.cv_fit_time, 4)))
        if seed is not None:
            random.seed(seed)

        # ratings must have 3 cols: users, items, ratings (in that order)
        train_set = data.build_full_trainset()

        self.algo = NMF(n_factors=n_factors, biased=self.biased,
             random_state=seed)

        if verbose:
            print('NMF Fitting...', end='')

        self.algo.fit(train_set)

        self.W = self.algo.pu
        self.H = np.transpose(self.algo.qi)

        # get predictions for *every* user/movie combo. These will be also
             compared to the actual ratings
        if verbose:
            print('Done')
            print('Generating all user-movie pairs for predictions...', end='')

        all_pairs = [(x, y, 0) for x in self.users.userID for y in
             self.movies.movieID]

        # getting predictions for ALL user/movie combos
        # took 40 seconds on 3.4 million rows
        if verbose:
            print('Done')
            print('Calculating predictions on all user-movie pairs...', end='')

        all_preds = self.algo.test(all_pairs)
        all_preds = pd.DataFrame([{'userID':y.uid, 'movieID':y.iid,
             'nmf_prediction':y.est} for y in all_preds])

        self.nmf_predictions = all_preds.merge(self.ratings,
             on=['userID','movieID'], how='left')
        self.nmf_predictions =
             self.nmf_predictions[['userID','movieID','rating','nmf_prediction']]
        self.trained_nmf = True

        if verbose:
            print('Done')

    def train(self, n_factors=15, run_cross_validation=True, seed=None,
         verbose=True):
        """[summary]

        Args:
            n_factors (int, optional): [description]. Defaults to 15.
            run_cross_validation (bool, optional): [description]. Defaults to
                 True.
            seed ([type], optional): [description]. Defaults to None.
```

```
            verbose (bool, optional): [description]. Defaults to True.
        """
        self.preprocess_tags(verbose=verbose)
        self.train_cosine_similarity(seed=seed, verbose=verbose)
        self.run_nmf(n_factors=n_factors,
            run_cross_validation=run_cross_validation, seed=seed,
            verbose=verbose)

    def get_similar_movies(self, movieID, number_of_movies=None,
        verbose=True):
        """[summary]

        Args:
            movieID ([type]): [description]
            verbose (bool, optional): [description]. Defaults to True.

        Raises:
            RuntimeError: [description]

        Returns:
            [type]: [description]
        """
        if not (self.preprocessed and self.trained_cosine):
            raise RuntimeError('Cannot make recommendations without training
                NMF, preprocessing, and training cosine first.')

        # get the index of the movie
        idx = np.where(self.movies_merged.movieID==movieID)[0][0]

        if verbose:
            print('Getting similar movies to ' +
                self.movies_merged.iloc[idx]['name'] + '...', end='')

        y = cosine_similarity(self.tfidf_matrix[idx], self.tfidf_matrix)
        idx_scores = pd.DataFrame([(idx, score) for (idx, score) in
            enumerate(list(y[0])) if score>0], columns=['idx','similarity'])

        result =
            pd.concat([self.movies_merged.iloc[idx_scores.idx].reset_index(),
            idx_scores], axis=1).sort_values(by='similarity', ascending=False)

        # get rid of transformed columns from movies_merged (except tag), and
            get the *original* name and genres with punctuation
        result.drop(columns=[x for x in [*self.movies_merged.columns,
            'index','idx'] if x != 'movieID'], inplace=True)
        result = result.merge(self.movies, on='movieID', how='left')
        result =
            result[['movieID','name','year','genre1','genre2','genre3','tags','similarity']]

        if verbose:
            print('Done')

        # don't include the movie we're finding similarities for
        if number_of_movies is not None:
```

```
        return result[1:].head(number_of_movies)
    else:
        return result[1:]

def get_recommendations(self, userID, number_of_recs=5, seed=None,
     show_user_likes=True, verbose=True):
    """[summary]
    Algorithm:
    1. Get 20 of the users' top ratings. Start with 5s, if > 20 exist,
        sample 20 randomly.
    2. If fewer than 20 5s exist, sample 4s until get to 20 (or use up all
        4s).
       - If there are no 5s or 4s, ignore the user's ratings, and just
         return the <number_of_recs> top predicted ratings for this user.
             Done.
    3. For each movie in the top list, calculate cosine similarity, and
        get the 10 most-similar
      movies which the user has NOT seen.
    4. Combine the 20 most-similar lists of 10 movies into a single list.
    5. Remove duplicates from this list, choosing the highest-similarity
        achieved
    6. For each movie, look up the predicted rating for this user.
    7. Multiply each movie's similarity times the predicted rating.
    8. Return the top <number_of_recs> predicted movies (or all if not
        enough). Done.

    Args:
        userID ([type]): [description]
        number_of_recs (int, optional): [description]. Defaults to 5.
        seed ([type], optional): [description]. Defaults to None.
        verbose (bool, optional): [description]. Defaults to True.

    Returns:
        pandas DataFrame: expected ratings. Columns: movieID, name,
            genres, weighted_rating
    """
    MAX_CONSIDERED_RATINGS = 20
    CONSIDER_N_SIMILAR = 10

    def combine_genres(df):
        # combine genres into a single column. Note that NaNs parse as
            float during apply
        df['genres'] = df.apply(lambda row: (row['genre1'] if not
            type(row['genre1'])==float else "") + \
                                        ("/" + row['genre2'] if not
                                            type(row['genre2'])==float
                                            else "") + \
                                        ("/" + row['genre3'] if not
                                            type(row['genre3'])==float
                                            else ""), axis=1)
        df.drop(columns=['genre1','genre2','genre3'], inplace=True)

    def get_subset_ratings():
        if verbose:
            print("Getting user's highest rated movies to start from...",
                end='')

        all_5s = self.ratings[(self.ratings.userID==userID) &
            (self.ratings.rating==5)]

        if len(all_5s) >= MAX_CONSIDERED_RATINGS:
            subset_ratings = all_5s.sample(MAX_CONSIDERED_RATINGS,
                random_state=seed)
        else:
            # use all 5s, and add in 4s until we have
                <MAX_CONSIDERED_RATINGS>
            subset_ratings = all_5s.copy()
            all_4s = self.ratings[(self.ratings.userID==userID) &
                (self.ratings.rating==4)]
            count_needed = MAX_CONSIDERED_RATINGS - len(all_5s)
            subset_ratings = pd.concat([subset_ratings,
                all_4s.sample(min(count_needed, len(all_4s)),
                random_state=seed)], ignore_index=True)

        subset_ratings =
            subset_ratings.merge(self.movies[['movieID','name']],
            on='movieID')

        if verbose:
            print('Done')

        return subset_ratings[['userID','movieID','name','rating']]

    def get_most_similar_movies(subset_ratings):
        if verbose:
            print("Finding similar movies to {} movies the user
                liked...".format(len(subset_ratings)))

        seen_movies =
            list(self.ratings[(self.ratings.userID==userID)].movieID)
        similar_movies = pd.DataFrame()

        for movie in subset_ratings.movieID:
            tmp_similar = self.get_similar_movies(movie, verbose=verbose)

            # limit to movies the user hasn't seen, and limit to top
                <CONSIDER_N_SIMILAR>
            tmp_similar =
                tmp_similar[~tmp_similar['movieID'].isin(seen_movies)].head(CONSIDER_N_SIMILAR)
            tmp_similar['similar_to'] =
                subset_ratings[subset_ratings['movieID']==movie].name.values[0]
            similar_movies = pd.concat([similar_movies, tmp_similar],
                ignore_index=True)

        # now remove duplicates, and get the top similarity for each movie
        similar_movies.sort_values(by='similarity', ascending=False,
            inplace=True)
        similar_movies.drop_duplicates(subset='movieID', keep='first',
        inplace=True)

    return similar_movies

if not (self.trained_nmf and self.preprocessed and
    self.trained_cosine):
    raise RuntimeError('Cannot make recommendations without training
        NMF, preprocessing, and training cosine first.')

if userID not in self.users.userID.values:
    raise ValueError('User {} does not exist in ratings dataset. If
        this is a new user, create a new user using the average
        ratings.'.format(userID))

if seed is not None:
    random.seed(seed)

review_counts =
    self.ratings[self.ratings.userID==userID].rating.value_counts()

if review_counts.get(5,0) + review_counts.get(4,0) == 0:
    # ignore user's ratings, and just get the user's top
        <number_of_recs> ratings
    if verbose:
        print("User has no ratings >= 4. Ignoring user's ratings,
            returning top predicted ratings.")

    # get only predicted ratings for ones the user hasn't seen
    subset_ratings =
        self.nmf_predictions.loc[(self.nmf_predictions.userID ==
        userID) & (self.nmf_predictions.rating.isna())].copy()
    subset_ratings = subset_ratings.merge(self.movies, on='movieID')
    combine_genres(subset_ratings)

    # add in columns that would have been calculated
    subset_ratings['similar_to'] = ""
    subset_ratings['similarity'] = np.nan
    subset_ratings['weighted_rating'] =
        subset_ratings['nmf_prediction']

    # reorder columns
    subset_ratings =
        subset_ratings[['movieID','name','year','genres','tags','similar_to','similarity','nmf_prediction','
    subset_ratings.sort_values(by='nmf_prediction', ascending=False,
        inplace=True)

    return subset_ratings.head(number_of_recs)

# get up to <MAX_CONSIDERED_RATINGS> 5s
subset_ratings = get_subset_ratings()

if show_user_likes:
    print('\n---------------\nHighest-reviewed movies for userID
        {}:'.format(userID))
```

```
            print(subset_ratings)
            print('\n---------------\n')

        # get the similarity for each movie in subset_ratings
        similar_movies = get_most_similar_movies(subset_ratings)

        # now we have the similarity scores for the movies most like the
            movies the user rated highest
        # get the predicted ratings, and multiply those by the similarity
            scores
        if verbose:
            print("Getting user's predicted ratings and calculated expected
                rating...", end='')

        user_predictions =
            self.nmf_predictions[self.nmf_predictions['userID']==userID]
        similar_movies = similar_movies.merge(user_predictions, on='movieID',
            how='inner')
        similar_movies['weighted_rating'] = similar_movies['similarity'] *
            similar_movies['nmf_prediction']

        if verbose:
            print('Done')
            print("Finalizing output...", end='')

        # combine genres and reorder columns
        combine_genres(similar_movies)
        similar_movies =
            similar_movies[['movieID','name','year','genres','tags','similar_to','similarity','nmf_prediction','weig
        similar_movies.sort_values(by='weighted_rating', ascending=False,
            inplace=True)

        if verbose:
            print('Done')

        return similar_movies.head(number_of_recs)
```

Appendix B. Use of Python package NMF_Cosine_Recommender.

```
import pandas as pd
from NMF_Cosine_Recommender import NMF_Cosine_Recommender as recommender

project_path = r'C:\Users\MyUser\Project\Data'

users = pd.read_csv(rf'{project_path}\users.csv')
movies = pd.read_csv(rf'{project_path}\movies.tsv', delimiter='\t')
ratings = pd.read_csv(rf'{project_path}\ratings.csv')
ml_tags = pd.read_csv(rf'{project_path}\movielens_20M_tags.csv')

# set up the NMF_Cosine_Recommender object
# don't include user and item bias in calculations
model = recommender(users, movies, ratings, ml_tags, biased=False)
```

```python
# train the model with 15 factors (i.e., rank = 15) in NMF
model.train(n_factors=15, run_cross_validation=False, seed=535)

# get recommendations for userID 37
recs = model.get_recommendations(userID=37, number_of_recs=20, seed=535)
print(recs)
```